Fast – Asymptotically Optimal – Methods for Determining the Optimal Number of Features*

Saied Tizpaz-Niari, Luc Longpré, Olga Kosheleva[0000−0003−2587−4209], and Vladik Kreinovich[0000−0002−1244−1650]

University of Texas at El Paso, El Paso, TX 79968, USA
{saeid,longpre,olgak,vladik}@utep.edu

Abstract. In machine learning – and in data processing in general – it is very important to select the proper number of features. If we select too few, we miss important information and do not get good results, but if we select too many, this will include many irrelevant ones that only bring noise and thus again worsen the results. The usual method of selecting the proper number of features is to add features one by one until the quality stops improving and starts deteriorating again. This method works, but it often takes too much time. In this paper, we propose faster – even asymptotically optimal – methods for solving the problem.

Keywords: Machine learning · Data processing · Optimal number of features · Asymptotically optimal method.

1 Formulation of the Problem

Selecting optimal number of features: an important problem. In machine learning – and in data processing in general – an important problem is selecting the number of features; see, e.g., [2].

- When we only use very few of the available features, the results are not very good – since we do not use a significant portion of the available information.
- As we increase the number of features, the results get better and better.
- However, at some point, we exhaust useful features, and we start adding features that practically do not contribute to the desired decision. In such situations, new features mostly adds noise, so the performance deteriorates again.

In other words, the efficiency E depends on the number of features n as follows:

* This work was supported in part by the National Science Foundation grants 1623190 (A Model of Change for Preparing a New Generation for Professional Practice in Computer Science), HRD-1834620 and HRD-2034030 (CAHSI Includes), EAR-2225395, and by the AT&T Fellowship in Information Technology.
It was also supported by the program of the development of the Scientific-Educational Mathematical Center of Volga Federal District No. 075-02-2020-1478, and by a grant from the Hungarian National Research, Development and Innovation Office (NRDI).

- the value $E(n)$ first increases with n,
- but at some point, it starts decreasing with n.

We need to find the value n_0 at which the effectiveness $E(n)$ is the largest.

How this problem is solved now. Of course, we can always do it by adding features one by one – and this is usually how people solve this problem now.

A better method is needed. In many cases – e.g., for machine learning – the adding-features-one-by-one algorithm is very time-consuming, since for each number of features, we need to re-train the neural network, and this takes time.

It is thus desirable to have faster methods for finding the optimal value n_0.

What we do in this paper. In this paper, we first describe a straightforward asymptotically optimal method for finding the optimal number of features. Then, we show how to further speed up the corresponding computations. Specifically:

- in Section 2, we formulate the problem in precise terms;
- in Section 3, we describe a straightforward asymptotically optimal method for solving this problem;
- in Section 4, we describe the new method, and we show that this method is indeed faster than the straightforward method.

2 Let Us Formulate the Problem in Precise Terms

What we are given. We are given a number N – the overall number of available features. We have an algorithm that:

- given a natural number $n \le N$,
- returns a real number $E(n)$.

What we know. We know that the function $E(n)$ first strictly increases, then strictly decreases. In other words, there exits some threshold value n_0 – that is not given to us – for which:

- if $n < n' \le n_0$, then $E(n) < E(n')$, and
- if $n_0 \le n < n'$, then $E(n) > E(n')$.

What we want to compute. We want to compute the threshold value n_0, i.e., the value at which the effectiveness $E(n)$ attains the largest possible value.

Usual method for computing n_0. The usual method for computing n_0 is trying $n = 1$, $n = 2$, etc., until we reach the first value n for which $E(n) < E(n-1)$. Then, we return $n_0 = n - 1$.

This method requires, in the worst case, N calls for the algorithm $E(n)$.

What we want. We want to come up with a faster method for computing n_0.

3 Straightforward Asymptotically Optimal Method for Solving the Problem

Main idea.

- For values $n < n_0$, we have $E(n) < E(n+1)$.
- For values $n \geq n_0$, we have $E(n) > E(n+1)$.

It is therefor reasonable to use bisection (see, e.g., [1]) to find the threshold value n_0.

Resulting method. At each iteration, we have values $n_- < n_+$ for which $E(n_-) < E(n_-+1)$ and $E(n_+) > E(n_++1)$. Based on the properties described in the previous paragraph, this implies that $n_- < n_0 \leq n_+$.

We start with the values $n_- = 0$ and $n_+ = N-1$. At each iteration, we take the midpoint

$$m = \left\lfloor \frac{n_- + n_+}{2} \right\rfloor$$

and check whether $E(m) < E(m+1)$. Then:

- If $E(m) < E(m+1)$, we replace n_- with the new value m.
- If $E(m) > E(m+1)$, we replace n_+ with the new value m.

At each iteration, the width of the interval $[n_-, n_+]$ is decreased by half. We stop when this width becomes equal to 1, i.e., when $n_+ - n_- = 1$. Once we reach this stage, we return n_+ as the desired value n_0.

How many calls to the algorithm $E(n)$ this method requires. We start with an interval $[0, N-1]$ of width $\approx N$. At each stage, the width of the interval decreases by a factor of 2. Thus, after k iterations, we get an interval of width $2^{-k} \cdot N$. The number k of iterations needed to reach the desired interval of width 1 can be therefore determined from the formulas $2^{-k} \cdot N = 1$, so $k = \log_2(N)$.

On each iteration, we call the algorithm $E(n)$ twice: to find $E(m)$ and to find $E(m+1)$. Thus, overall, this method requires $2 \cdot \log_2(N)$ calls to the algorithm $E(n)$.

This method is asymptotically optimal. We need to find a natural number n_0 from the interval $[0, N]$. In general, by using b bits, we can describe 2^b different situations. Thus, the amount of information b that we need to determine n_0 must satisfy the inequality $2^b \geq N$, i.e., equivalently, $b \geq \log_2(N)$. To get each bit of information, we need to call the algorithm $E(n)$. Thus, to find n_0, we need to make at least $\log_2(N)$ calls to this algorithm.

The above algorithm requires $2 \cdot \log_2(N) = O(\log_2(N))$ calls. Thus, this method is indeed asymptotically optimal.

Natural question. The straightforward method described in this section is asymptotically optimal, this is good. However, still, this method requires twice more calls to the algorithm $E(n)$ that the lower bound. Thus, a natural question is: can we make it faster?

Our answer to this question is "yes". Let us describe the new faster method.

4 New Faster Method: Description and Analysis

Preliminary step. First, we compute the value $E(m)$ for the midpoint m of the interval $[0, N]$, so we form an interval $[n_-, n_+] \stackrel{\text{def}}{=} [0, N]$.

Iterations. At the beginning of each iteration, we have the values $n_- < n_+$ for which:

- we know the values $E(n_-)$, $E(n_+)$ and $E(m)$, where m is the midpoint of the interval $[n_-, n_+]$, and
- we know that $E(n_-) < E(m) > E(n_+)$.

We stop when $n_+ - n_- = 2$, in which case we return $n_0 \stackrel{\text{def}}{=} n_- + 1$.

At each iteration, we first select, with equal probabilities 0.5, whether we start with the left subinterval or with the right subinterval.

If we start with the left subinterval, then we compute the midpoint L of this subinterval, and compute the value $E(L)$. Then:

- If $E(L) > E(m)$, i.e., if $E(n_-) < E(L) > E(m)$, then we replace the interval $[n_-, n_+]$ with the new half-size interval $[n_-, m]$. In this case, the iteration is finished. So, if the stopping criterion is not yet satisfied, we start the new iteration.
- On the other hand, if $E(L) < E(m)$, then we compute the midpoint R of the right subinterval $[m, n_+]$, and compute the value $E(R)$. Then:
  - If $E(m) > E(R)$, i.e., if $E(L) < E(m) > E(R)$, then we replace the interval $[n_-, n_+]$ with the new half-size interval $[L, R]$.
  - On the other hand, if $E(m) < E(R)$, i.e., if $E(m) < E(R) > E(n_+)$, then we replace the interval $[n_-, n_+]$ with the new half-size interval $[m, n_+]$.

If we start with the right subinterval, then we compute the midpoint R of this subinterval, and compute the value $E(R)$. Then:

- If $E(m) < E(R)$, i.e., if $E(m) < E(R) > E(n_+)$, then we replace the interval $[n_-, n_+]$ with the new half-size interval $[m, n_+]$. In this case, the iteration is finished. So, if the stopping criterion is not yet satisfied, we start the new iteration.
- On the other hand, if $E(m) > E(R)$, then we compute the midpoint L of the left subinterval $[n_-, m]$, and compute the value $E(L)$. Then:
  - If $E(L) > E(m)$, i.e., if $E(n_-) < E(L) > E(m)$, then we replace the interval $[n_-, n_+]$ with the new half-size interval $[n_-, m]$.
  - On the other hand, if $E(L) < E(m)$, i.e., if $E(L) < E(m) > E(R)$, then we replace the interval $[n_-, n_+]$ with the new half-size interval $[L, R]$.

Why this algorithm works. If for some values $n_- < m < n_+$, we have $E(n_-) < E(m) > E(n_+)$, then:

- we cannot have $n_0 \le n_-$, since then $n_- < m$ would imply $E(n_-) < E(m)$; thus, we must have $n_- \le n_0$; and

– we cannot have $n_+ \le n_0$, since then $m < n_+$ would imply $E(m) < E(n_+)$; thus, we mist have $n_0 \le n_+$.

Thus, in this case, we must have $n_0 \in [n_-, n_+]$.

How many calls to the algorithm $E(n)$ this method requires. Each iteration reduced the width of the original interval $[n_-, n_+] = [0, N]$ by half. So, similarly to the straightforward algorithm, we need $\log_2(N)$ iterations

On each iteration, we first make the first call to $E(n)$ and then the first comparison. We have two possible results of this comparison, so it is reasonable to assume that each comparison result occurs with probability 0.5. So, on each iteration:

– with probability 0.5, we require only one call to the algorithm $E(n)$, and
– with probability 0.5, we require two calls.

Thus, the expected number of calls on each iteration is $0.5 \cdot 1 + 0.5 \cdot 2 = 1.5$.

Since we make an independent random selection on each iteration, the numbers of calls on different iterations are independent random variables. Thus, due to the large numbers theorem (see, e.g., [3]), the overall number of calls will be close to the expected number of calls, i.e., to $1.5 \cdot \log_2(N)$. This is clearly faster than the number of calls $2 \cdot \log_2(N)$ required for the straightforward algorithms.

References

1. Th. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts, 2022.
2. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*, MIT Press, Cambridge, Massachusetts, 2016.
3. D. J. Sheskin, *Handbook of Parametric and Nonparametric Statistical Procedures*, Chapman and Hall/CRC, Boca Raton, Florida, 2011.